Type Abstraction in Formal Protocol Specifications with Container Types

Joachim Thees

University of Kaiserslautern, Postfach 3049, D-67653 Kaiserslautern, Germany,
thees@informatik.uni-kl.de

Abstract. In this paper, we propose a seamless integration of the concept of *"(universal) container types"* into Formal Description Techniques (FDTs), which introduces a new concept of data abstraction. We show how this syntactical and semantic extension increases the expressiveness in the area of component reuse, without sacrificing the formal precision of the FDT. The ideas are exemplified for Estelle, but apply to other FDTs in the protocol domain (e.g., SDL) as well. Furthermore we will demonstrate, how this extension increases the capability to formally specify the static and even the dynamic packet composition and decomposition of pre-existing protocols (like TCP/IPv6 or XTP).

1 Introduction

Type safety is an important requirement to the formal (i.e. mathematically precise) specification of communication systems. Therefore all data handling operations must have an implementation independent and unique semantics. However, the level of type safety enforced by many FDTs in the protocol domain (e.g., Estelle [1, 2] and SDL [4]) has a negative impact on the complexity and usability of large, hierarchically structured, and heterogeneous protocol stacks, especially against the background of reuse approaches of protocol components (e.g., as open systems with SDL or Open Estelle [10]).

In this paper, we will demonstrate why this level of type safeness hampers a generic and application independent definition of communication protocols and services. This limitation becomes apparent especially at very complex, heterogeneous specifications and against the background of reuse approaches of protocol components (e.g., as open systems with SDL or Open Estelle [10]).

As a solution to this problem, we propose a seamless integration of the concept of *"(universal) container types"*, which introduces a new concept of data abstraction, but avoids the overhead of a complete data description technique like ASN.1 [3]. We show how this syntactical and semantic extension increases the expressiveness in the area of component reuse, without sacrificing the formal precision of the FDT. The ideas are exemplified for Estelle, but apply to other FDTs in the protocol domain (e.g., SDL) as well.

H. König, M. Heiner, and A. Wolisz (Eds.): FORTE 2003, LNCS 2767, pp. 383–398, 2003.

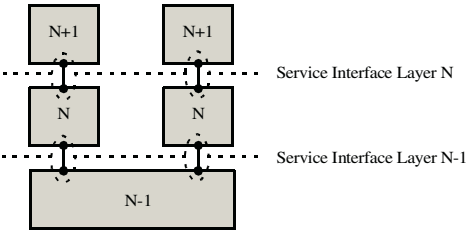

Fig. 1. Layered model for structured communication systems

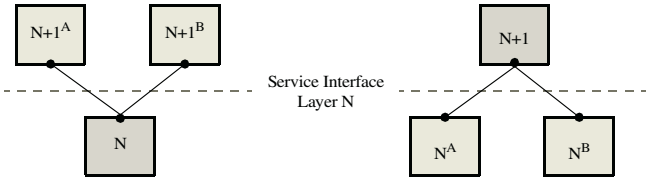

Fig. 2. Independence of service users and service providers

1.1 Hierarchical Communication Systems

Service hierarchies are an important means for the structuring of communication systems. A system is structured into a number of layers (see Fig. 1), which provide services to higher layers by making use of the services of lower layers. The service interface of these layers hides the internal details of each layer from all other layers. So these interfaces can be restricted to service aspects only. This concept becomes even more important, if different service users access the same service (see left hand side of Fig. 2) or a service user shall be bound to different service providers without modifications (see right hand side of Fig. 2). Such situations are very common in real world communication systems. For example, we are used to running different applications on our workstation with the same TCP/IP protocol implementation without any system reconfiguration. On the other hand, the TCP/IP implementation on a workstation should not depend on the kind of basic technology that is used to access the internet, especially if different basic technologies are used concurrently.

Obviously the concept of information hiding and the abstraction provided by minimal service interfaces are very important requirements for heterogeneous structured communication systems.

1.2 Data Representation in Hierarchical Communication Systems

The OSI Reference Model [5] defines access to data transfer services in terms of Service Access Points (SAPs), which allow a service user (layer N+1) to pass its payload as a Service Data Unit (N-SDU) to a service provider (layer N, see Fig. 3). The protocol machine of layer N will now create a Protocol Data Unit

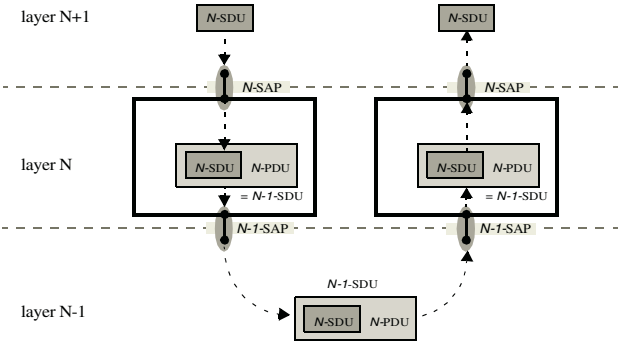

Fig. 3. Nesting of SDUs into PDUs

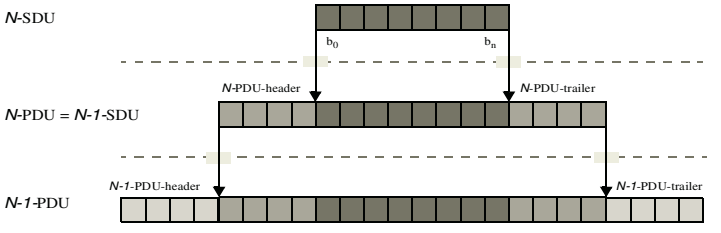

Fig. 4. Framing of SDUs into PDUs at byte encoding level

(N-PDU) that includes the N-SDU (i.e. the initial payload) and some additional protocol specific data. This N-PDU itself is now used as a N-1-SDU for the SAP to the next lower service (layer N-1). As soon as this service at layer N-1 delivers the N-1-SDU to the peer protocol machine, the payload (N-SDU) is extracted and transferred to the layer N+1 service user.

This mechanism can be found in most layered communication systems which honor the concept of separation of concerns between the layers: the payload of any service user is handled and transmitted by the lower service providers basically without any interpretation of its contents.[1] Therefore at implementation level payloads (SDUs) are most often represented as unstructured byte sequences and the inclusion of an SDU into an PDU is handled as a simple concatenation of byte sequences (header, payload and trailer), known as *framing* (see Fig. 4).

1.3 Formal Specification of Hierarchical Communication Systems

A natural representation of services and protocol machine structures in Estelle is based on modules. In Fig. 1 we have seen a typical module instance hierarchy implementing a protocol stack and its service structure. This is at first glance

[1] Some integrated protocols like TCP and IP share some data (e.g., destination IP-Address) inside the payload between different layers, but these could also be passed redundantly as payload and separate service parameter.

appropriate to our information hiding requirements, since Estelle modules have a well defined external interface and encapsulate all further internal details from their environment.

But as soon as we start to model the internal aspects of the protocol machines, we will have to specify the framing and un-framing mechanisms described above. The type safeness of Estelle implies only one precise way for doing this: On every layer (N) the SDU (N-SDU) is embedded into the PDU (N-PDU) as a record-component.[2] Framing is implemented by assigning SDU value to the appropriate PDU component under construction. The resulting N-PDU is already the N-1-SDU and can transferred (also type safe[3]) by the underlying communication service N-1. At the destination module instance of layer N, the un-framing is implemented by direct access to the N-SDU component of the N-PDU record.

The advantage of this approach is obvious: All data transfers and assignments are type safe across the whole system. The disadvantages become apparent as soon as we consider systems with a great number of framing layers: since for every layer N the N-PDU aggregates its N-SDU (i.e. the N+1-PDU), the data structures become more and more complex with every layer we step down in the hierarchy (see Fig. 5). Obviously there is some kind of "inheritance" of payload types down the protocol stack.[4] This is even more undesirable as it acts inversely compared to the service complexity[5] provided at the individual layers.

The solution appears even worse if we consider heterogeneous protocol stacks, where all services may be used by different protocols or applications (see Fig. 2). Except of the pathological case, where all users of a service N use exactly the

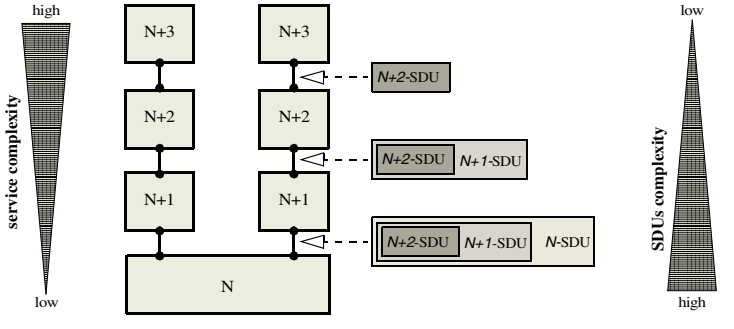

Fig. 5. Complexity of Services and SDUs in Communication Hierarchies

[2] Alternatively also an array over N-SDU-types could be used, but in most cases this is only useful if the array is part of an N-PDU-Record, which also holds additional protocol specific data for N.

[3] The type safeness of the system can be proved by structural induction over the number of layers.

[4] We call this composing of types "vertical" (orthogonal to the "horizontal composition" noted below).

[5] In most cases we put a service (or protocol) on top of another service in order to improve it in any way.

same N-SDU (i.e. payload) type, a handling of different N-SDUs is necessary. The obvious solution is to embed not only one N-SDU type into the N-PDU, but all possible N-SDU (i.e. payload) types, e.g., as variants of a variant record.[6] Since heterogeneous protocol applications can occur on every layer and the resulting variant records are also "inherited" to the external interfaces of all service interfaces below, the combination of horizontal and vertical composition results in a real type explosion of exponential complexity.[7]

1.4 Data Type Abstraction

Obviously the kind of type safe SDU-PDU-nesting shown above is only useful for protocol specifications with a low number of layers and very little heterogeneity. But this requirement is contradictory to the approach of true generic service and protocol machine specification. Especially if we try to specify communication services separately from and independently of concrete service users (e.g., as an open system with the Estelle extension Open Estelle [10]) we would have to specify the payload types of all possible[8] users in advance.

On the other hand, the internal structure of the payloads is of no interest for most data transfer services, since they simply transmit them without any interpretation.[9] Consequently the internal structure of the PDU of a protocol machine could be kept as one of its internal secrets, which are isolated from the environment in terms of the external interface.

Basically we are looking for a data type abstraction for SDU and PDU types in formally specified protocol machines for the following tasks:

1. SDU types of services (and also of protocols) should be abstract, as long as only the service users are interested in their concrete structure and contents. This allows us to give an abstract service specification without reference to a concrete payload type. Furthermore, the independence from any concrete payload type also proves automatically, that the service is defined independently of the concrete payload type.[10]
2. Protocol machines should be able to keep their concrete PDU types as an internal secret, as long as lower services do not have to consider their content. This simplifies the external interface, since it can reflect only service

[6] We call this composing of types "horizontal" (orthogonal to the "vertical composition"noted above).

[7] In a system with n service layers and m variants on every layer we have a PDU type with $O(m^n)$ at the lowest service provider.

[8] For a truely generic service (e.g. specified as open system in Open Estelle) any not pointer containing type would have to be supported.

[9] In implementation level services payload is most often represented and handled as simple byte sequence, where only the number of bytes and abstract properties like their checksum are considered.

[10] In Fig. 5 service provider N knows the complete structure of the payload (N-SDU) and therefore could access or even modify any substructure from higher layers (e.g., the N+2-SDU). In most cases this is not intended and therefore should be avoided at conceptional level.

aspects and can be kept free of internal details. For example, the external interface of a transport layer protocol machine should not depend on protocol mechanisms like acknowledgement strategies or sliding window lengths.

3. Service users (including protocol machines) should be able to pass their payload types to the abstract SDU-parameters of these abstract service providers and protocol machines. Of course this step must be reversible at the receipt of SDU-parameters by the peer service users.
4. All operations must be semantically formal, i.e. type safe.

Unfortunately, Estelle has no means to do this kind of data abstraction, especially if we take into consideration the demand to formally specify services with Open Estelle completely independent of their later users:

- The SDU-type polymorphism already discussed above is only applicable if all service users are known in advance, and even then it is only useful for a small number of type variants.
- The use of *incomplete type specifications* (e.g., "`TYPE T = ... ;`") makes the whole specification incomplete, i.e. it has no formal semantics.
- Any kind of *type parametrization*[11] of formally described open systems is only useful for a single SDU type, but if several service users with different SDU-types access the same service provider instance in the same system, the resulting system suffers from the same problems (see left hand side of Fig. 2).
- The application of *primitive encoding and decoding functions* as proposed in Annex B of [2] represents all SDU types as byte arrays of fixed size. Since the size of these arrays has to be specified and therefore limited in advance, these SDU types do not lead to a truly universal service definition.[12] Apart from this, the abstraction level of specifications using this technique is only appropriate if we have to consider encoding specific aspects, which is far below the goals of Estelle as a formal description technique.

The introduction of data type description languages like *ASN.1* [3] into Estelle requires a close coupling and interaction of both, the Estelle type description and the ASN.1 type descriptions. This goes beyond our goals described above, since ASN.1 aims especially to the exchange of data between different, heterogeneous worlds.

In the following section we will present a very simple and fully embedded solution for the data abstraction problem described above.[13]

[11] The concept of *type parametrization* is not part of Standard Estelle.

[12] Obviously, for every number of bytes in this array, we can find a data type that cannot be encoded into it in all cases.

[13] The data serialisation mechanism of Java doesn not provide a true container type according to Def. 1, since the conversion is not implicit. Another (also not universal) container type candidate in Java is the class Object that is a base class for all non-primitive types in Java.

2 Universal Container Types

As a solution for the data abstraction problems in hierarchical communication systems we will now introduce the concept of "*(Universal) Container Types*" to Estelle. We will start with some language independent considerations.

Definition 1. *Container Type*
A Type TX is called **Container Type** *of Type T if*
TX and T are assignment compatible and for all Values x of type T is valid: if x is converted to type TX and then back again to type T, then both conversions are allowed and the resulting value is equal to x.

The idea of this definition is that a container type TX of a type T can hold any value of type T without loss. For example, the Estelle type INTEGER is a container type for the INTEGER sub-range type T=0..9. On the other hand, T=0..9 is *not* a container type for INTEGER, since the value 10 can't be converted legally to type T. Furthermore, every type T is its own (trivial) container type.

For any pair of types with $val(TX) \supseteq val(T)$, TX is a container type for T, but the definition it is not restricted to these situations. For example, with an implicit conversion rule for integers to character strings (e.g., in hex-representation) and vice versa, character strings could serve as container types for integers.

A *universal container type* is a type that has the ability to serve as a container type for any type. Some languages, which support the concept of data serialization (e.g., into character strings in Java), may have universal container types.

2.1 Estelle Extension "Universal Container Type"

Obviously Estelle does not have a universal container type, i.e. a type that can serve as a container type for any Estelle type T. Hence we will now introduce a new Estelle type "*any-type*", which extends the Estelle syntax given in [2]:

Definition 2. *Syntax of the Estelle Container Type "any-type"*
any-type = "`ANY`" "`TYPE`" .
type-denoter = — | *any-type* .

With this extension we can use the non-terminal[14] *any-type* as a new type-denoter for the definition of variables, parameters, new types, etc., similar to a predefined type name. Most interesting is its application as a interaction parameter in the channel definition below:

Example 1. Syntactical Use of any-type

[14] We use the combination of Standard-Estelle keywords "`any`" and "`type`" to avoid the introduction of a new keyword. This simplifies the integration of our extension into existing specifications, since no collisions with identifiers are possible. Furthermore the term "`any type`" mimics Estelle constructs like "`any integer`".

```
TYPE T = ANY TYPE;
VAR x: ANY TYPE;
PROCEDURE f(x: ANY TYPE); BEGIN (* ... *) END;
CHANNEL ChService_N(User,Provider);
        BY User:     D_Send(Data: ANY TYPE);
        BY Provider: D_Recv(Data: ANY TYPE);
```

The syntactical non-terminal *any-type* denotes a new type (also called "*any-type*" here). As already suggested by it's name, this new type has special properties concerning its compatibility with other types. These properties are defined by the following extension of the definition of assignment compatibility given in section 6.4.6, Annex C[15] of [2]:

Definition 3. Extended Type Compatibility *of the any-type extension*
A value of type T2 shall be designated **assignment-compatible** *with a type T1 if any of the following seven statements is true:*

a .. e) { *unmodified* }
f) *T1 is the any-type and T2 is not pointer-containing.*
g) *T1 is not pointer-containing and T2 is the any-type.*

At any place where the rule of assignment-compatibility is used

a, b) { *unmodified* }
c) *it shall be an error, if T1 is the any-type and the value of T1 was not created by conversion from type T2 to the any-type.*

{ *rest unmodified* }

Clause (f) means, that assignments to the *any-type* are allowed from all types, which are not pointer containing.[16] As a consequence, the *any-type* itself is also not pointer containing.[17] This restriction is a requirement to use the *any-type* as parameters to interactions (see channel definition in example 1).

This syntactical extension already fulfills our requirements for abstract communication services in hierarchical communication systems. The SDU type *any-type* with its assignment compatibilities allows us to specify *really abstract service interfaces*. Any service user can convert its specific payload type into an *any-type* and pass it as its SDU to the abstract service provider (e.g., N-1 in Fig. 1). The service provider passes this *any-type* SDU without further consideration of its possible contents to the destination service user (still as *any-type* SDU). This destination service user can convert the *any-type* value into its original type and value and handle its contents appropriately.

[15] Annex C defines the Pascal-subset contained in Estelle. Obviously the proposed extension could also be applied to native imperative languages like Pascal.

[16] Obviously the *any-type* is not really a universal container type. But it is in fact universal for the set of types that can be passed as interaction parameters between module instances (i.e. all not pointer containing types).

[17] It is not pointer containing *by value*. Technically being no real structured type, the definition of pointer containing in [ISO97] is not appropriate for this type, since it refreres only to its *syntactical structure*.

Apparently we attenuate the type safeness of Estelle with this kind type conversions into the *any-type* and back. This is only acceptable, if the semantics of the conversions is unambiguous in a mathematical sense. This is achieved by the following two requirements:

1. Besides assignment and the conversions defined in Def. 3, there are no operations or actions on instances of the *any-type*.
2. If there is a conversion from an arbitrary type to the "*any-type*", then the resulting *any-type* value can only be converted back to the initial type (see clause (c) in Def. 3).

So the type safeness of the possible operations with *any-type* is secured by construction. Obviously an important requirement to the assignment compatibility, clause (c) in Def. 3, can only be verified at runtime. This is not a new concept to Estelle, since e.g., also the correctness of pointer values or variant records cannot be verified statically (see Sections 6.4.3 and 6.4.4 of Annex C of [2]). We will come back to this aspect in Sections 3 and 4.

The semantics of the proposed Estelle extension is based directly on the definition of container types (see Def. 1):

Definition 4. Semantics *of the any-type*
The any-type is a container type for all not-pointer-containing types.

Consequently, the use of the *any-type* has a precise semantics and a specification does *not* become "not well-formed" because of its use (instead of e.g., the type denoter "..."; see section 8.2.3 of [2]).

2.2 Application of the Estelle Extension "Universal Container Type"

For the demonstration of the practical application of the proposed *any-type* extension, we will now implement the essential aspects of the communication scenario shown in Fig. 6. Two service users communicate through an abstract and application independent service interface based on the *any-type* as SDU types (see example 2).

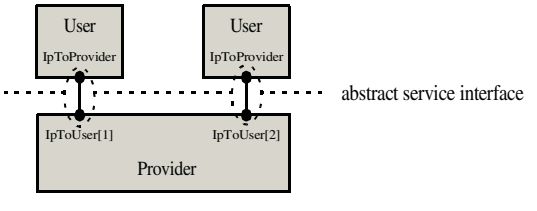

Fig. 6. Module and Connection Structure of Example Communication Scenario

Example 2. Abstract Service Interface

```
TYPE Td = ANY TYPE;
CHANNEL ChService_N(User,Provider);
        BY User:      D_Send(Data: Td);
        BY Provider:  D_Recv(Data: Td);
```

With this definition of Td the service user can send any payload (here: local type T1) without explicit conversion as interaction parameter:

Example 3. Service User (Sender)

```
IP IpToProvider: ChService_N(User);     (* external IP *)
(* ... *)
TYPE T1 = RECORD
            a: REAL;
            b, c: BOOLEAN;
            END;
VAR x1: T1;
TRANS
   BEGIN
   OUTPUT IpToProvider.D_Send(x1); (* any-type conversion *)
   END;
```

In the opposite direction the any-type parameter received with the `D_Recv` interaction can be converted back to its original type, in order to evaluate its contents:

Example 4. Service User (Receiver)

```
TRANS
   WHEN IpToProvider.D_Recv(Data: Td)
      VAR x: T1;
      BEGIN
      x := Data;            (* any-type back-conv. *)
      (* evaluates x ... *)
      END;
```

As in example 4, another service user may transfer completely different payload types through the same service and the same interaction parameter. But it is important that the *any-type* value sent is finally received by the right receiver in order to be converted back to the right (i.e. the sent) type.

The most interesting part finally is the service provider, which becomes completely independent of any possible service user. The *any-type* payload is simply handled without any knowledge of its specific contents:

Example 5. Service Provider

```
IP IpToUser: ARRAY [1..2] OF ChService_N(Provider); (* ext. IPs *)
(* ... *)
TRANS
   ANY i: 1..2 DO
      WHEN IpToUser[i].D_Send(Data: Td)
         BEGIN
         OUTPUT IpToUser[3-i].D_Recv(Data);
         END;
```

Obviously this kind of service abstraction can be used on all service levels of a hierarchical communication systems (see e.g., Fig. 1). Like the user modules in the example above, also the protocol machines can hide their specific PDU structure internally.

A very interesting result of this approach is that finally the SDU types of all service interfaces at all layers may be identical (i.e. *any-type*) and therefore even some or all service interfaces could be defined identically.[18] This proofs, how far this approach reduces the service interfaces and the external interfaces of protocol machines to aspects induced only by core requirements.

3 Dynamic Aspects of Container Types in Communication Systems

An important requirement to the correctness of a conversion from an *any-type* value back to a regular Estelle type is the conformity of this destination type with the initial type, which created the *any-type* value (see clause (c) in Def. 3). We will now discuss, how this correctness can be ensured in more complex situations.

Therefore we will first have a look at manual implementations of data type handling in existing ("*real world*") protocols.

3.1 Data Type Handling in Real World Protocol Implementations

In real world communication systems (e.g., TCP/IP) several protocols have to share the same basic data transport services. As we have already seen in Section 1.2, these data transport services represent SDUs as unstructured byte arrays (frames).

If such a data frame is received from the lower data transport service, its internal structure must be *reconstructed*. This means, the data must be reinterpreted from an unstructured data object to a (more) structured one. Obviously this is also exactly the idea behind a cast from an *any-type* to a more specific type.

But how can we assign a structure (i.e. a type) to the yet unstructured data frame? If we can expect only one type of data frames with always the same structure, we can simply assign it to this type. E.g., if our service users send exclusively single two-byte little-endean integers to the data transport service and we assert a faultless service, we can also expect that every frame received contains only such integers.[19]

This shows the basic idea of the transfer of structured data as unstructured SDUs: on receipt we reassign a structure to the unstructured data based on our *a priori knowledge* (or at least our *expectation*) of its original type. If our

[18] Of course, this can only happen if possible additional service specific parameters (e.g., destination addresses or q.o.s. parameters) are identical, too.

[19] Obviously in this example we could at best check the size of these frames (each should be 2 bytes long).

expectation is correct, we will be able to reconstruct the data sent correctly. If the data received is of different type than expected, we will fail: at best case we will be able to detect this mismatch at means of other assertions (e.g., wrong frame size, illegal contents, wrong checksums), but if not, we will reconstruct and handle wrong data (i.e. data never sent). This is a basic property of any communication over a not strictly type safe communication systems.

So finally the assignment of the correct type to a yet unstructured piece of data is a basic task in all real communication systems. The container type extension just introduces this idea into formal protocol specifications: The assignment of the correct type to a unstructured piece of data is modeled by the cast of an *any-type* value to a more specific type. And analogous to the real world implementations, the correctness of this type assignment is left to the specifier and cannot be checked statically.[20]

3.2 Successive Data Decomposition

We have seen that we have to assign the right structure (i.e. type) to a unstructured data received from our communication system. But what if there may be different data types that are transmitted over the same service, i.e. the service is multiplexed? In this case all known protocols use frame types that have a common substructure for all variants. These common parts are in most cases placed at beginning of the frame (the so called *header*).

Since the structure of this common part is known a priori, we can reconstruct its contents without knowledge of the complete structure of the whole frame. In this common structure we will find additional parameters for the further type assignment of the data object. So we can reconstruct the complete type structure of frame step by step.

We illustrate this idea by means of the TCP/IPv4 protocol [6]. All TCP/IP frames start with a common IP header, followed by some payload data (Fig. 7).

Inside of this common header we find a field named "Protocol", which (partially) identifies the substructure of the data field: A value of 6 indicates a TCP data field [7], a value of 17 indicates a UDP data field [8].

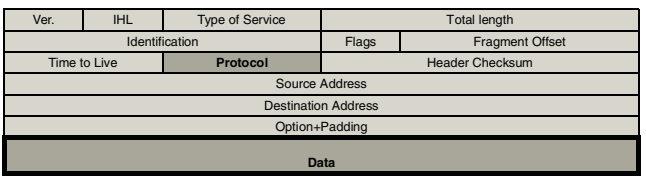

Fig. 7. IPv4 PDU structure

[20] There may be consistency checks, which detect wrong type casts in implementation or simulations (see Section 4). In some cases also a full static check may be possible (e.g. in simple systems, where only one type is converted into the *any-type*).

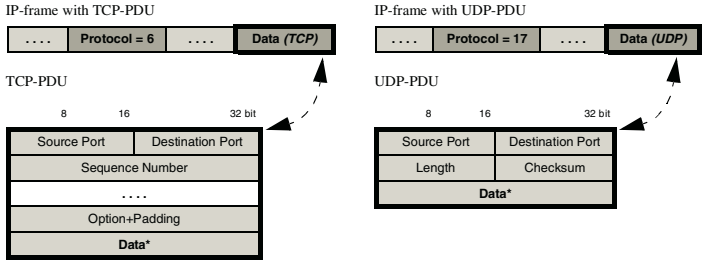

Fig. 8. TCP- and UDP-PDUs embedded into IPv4-PDU

In order to decomposed a IPv4 frame received from a lower service, we will first have to assign the a priori known structure to the IP header with its protocol field. The Data field of the Frame is kept unstructured, yet. In a second step we can start to assign a structure to the IPv4 Data field, depending on the Protocol value (see Fig. 8).

With this knowledge of the TCP or UDP structure we can further detect the Destination Port, pass the nested Data part to the bound application and continue its decomposition with application level knowledge of its structure, and so on.

Obviously we are practicing some kind of (vertical) successive data decomposition: Starting from the a priori known structure of a frame we assign types and decompose the frame step by step.

In order to model this kind of decomposition with our container type extension, we have to create a nested data structure with a structural known part and an unknown part. The known part (e.g., the frame header) is modeled with regular Estelle types, whereas the unknown part is an *any-type*. Both parts are aggregated into a record (see type `IPv4_Frame` in example 6).

Example 6. Type Hierarchy for Successive Data Decomposition

```
TYPE Abstr_Frame = ANY TYPE;
TYPE IPv4_Header = RECORD
                     (* ... *)
                     Protocol: 0..255;
                     (* ... *)
                   END;
TYPE IPv4_Frame =  RECORD
                     Header: IPv4_Header;
                     Data:   ANY TYPE;
                   END;
TYPE TCP_PDU = ... ;
TYPE UDP_PDU = ... ;
```

If we receive a frame (modeled as an *any-type*, since the lower service provider does not know anything about the frame structure), we first convert it into the type `IPv4_Frame`, based on our a prior knowledge about its structure (see

example 7). Now we can access the header with its Protocol field. Then we can continue the successive data decomposition for any known value of Protocol (here: 6 and 17 for TCP and UDP).

Example 7. Successive Data Decomposition for IPv4

```
TRANS
  WHEN FromLowerService.Frame{atf: Abs_Frame}
    VAR f: IPv4_Frame;
        tpdu: TCP_PDU;
        updu: UDP_PDU;
      BEGIN
        f := atf;                              (* type known a priori *)
        IF f.Header.Protocol = 6 THEN
          BEGIN
          tpdu := f.Data;                (* type for Protocol 6: TCP *)
          (* ... *)
          END
        ELSE IF f.Header.Protocol = 17 THEN
          (* ... *);
      END;
```

The composition of such a frame has to be done exactly in the opposite way, in order to make every later decomposition step well defined (see Section 2.1).

Finally we will demonstrate the potential of our approach at the specification of truly dynamic data types, as they are used in many modern protocols. In protocols like IPv6 [9] or XTP 4.0 [13], frames are no longer of fixed structure with a single, variable payload part, but instead consist of several parts, which are optional, repetitive or of variable size. These so called segments are chained inside the packet, starting from a (once again a priory known) header of fixed type (see Fig. 9).

Standard Estelle has no appropriate means to specify such highly dynamic frame structures. With the container type extension we can use a variable of type *any-type* to hold any intermediate state of the packet composition (see Fig. 10). Starting with the last segment (which is directly assigned to the *any-type* variable), we insert step by step one segment, by creating an auxiliary record value with a component of the current segment type and an *any-type* component for the already constructed rest of the frame (like `IPv4_Frame` in example 6). With this method we can construct segment sequences of arbitrary structure.

The construction process is done in this "reverse order", because the decomposition process has to be done the same way, but in opposite order, starting with the forefront (i.e. a priori known) header.

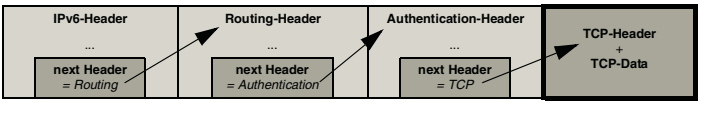

Fig. 9. TCP-PDU with different segments of a IPv6-PDU

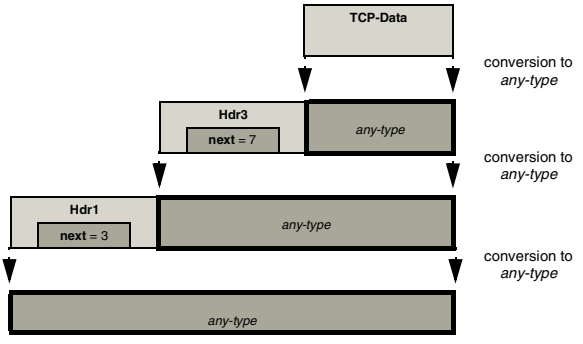

Fig. 10. Successive Data Composition *any-type*-Aggregation

4 Implementation Issues

Besides a basic implementation approach for the implementation of the *any-type* on top of not object oriented implementation platforms, we have also implemented it with our C++ based Estelle compiler XEC [11, 12]. Because of its object oriented data model, the container type extension could be integrated easily. By the use of class templates for the internal abstraction of the contained data objects and `dynamic_cast`-operators for the casts from an *any-type* value back to a specific type, a full dynamic type check is provided at implementation level.

5 Summary

We have shown how the level of type safety enforced by many FDTs in the protocol domain (e.g., Estelle and SDL) has a negative impact on the complexity and usability of large, hierarchically structured, and heterogeneous protocol stacks, especially against the background of reuse approaches of protocol components (e.g., as open systems with SDL or Open Estelle).

As a solution to this problem, we proposed the concept of "(universal) container types" and their seamless integration into FDTs, which introduced a new concept of data abstraction, but avoids the overhead of a complete data description technique like ASN.1. We further demonstrated how this syntactical and semantic extension increases the expressiveness in the area of component reuse and flexible data representation, without sacrificing the formal precision of the FDT. The ideas were exemplified as the "any-type"-extension for Estelle, but apply to other FDTs in the protocol domain (e.g., SDL) as well.[21]

[21] This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) under grant Go 503/4-2.

References

1. Dembinski, P., Budkowski, S.: Specification Language Estelle, in: M. Diaz et al. (eds.), "The Formal Description Technique Estelle", North-Holland, 1989
2. ISO/TC97/SC21: Estelle - A Formal Description Technique Based on an Extended State Transition Model, ISO/TC97/SC21, IS 9074, 1997
3. ITU-T Recommendation X.680 – X.683 (07/02): Abstract Syntax Notation One (ASN.1), International Telecommunication Union (ITU), 2002
4. ITU-T: CCITT Specification and Description Language (SDL), Recommendation Z.100 (03/03), 1994
5. ISO/TC 97/SC 16, ISO 7498, "Data Processing – Open Systems Interconnection – Basic Reference Model", 1981
6. J. Postel (Edt.), "Internet Protocol, Specification", RFC791, 1981
7. J. Postel (Edt.), "Transmission Control Protocol, Specification", RFC793, 1981
8. J. Postel (Edt.), "User Datagram Protocol, Specification", RFC768, 1980
9. S. Deering, R. Hinden (Edts.), "Internet Protocol, Version 6 (IPv6), Specification", RFC1883/2460, 1995
10. J. Thees, R. Gotzhein: Open Estelle - An FDT for Open Distributed Systems, in: S. Budkowski et.al. (Edts.): Formal Destcription Techniques and Protocol Specification, Testing and Verification (FORTE XI/PSTV XVIII'98), Kluwer Academic Publishers, Boston/Dordrect/London, 1998
11. J. Thees: The eXperimental Estelle Compiler - Automatic Generation of Implementations from Formal Specifications, in: Formal Methods in Software Practice (FMSP'98), Clearwater Beach, Florida, USA, 1998
12. J. Thees: Protocol Implementation with Estelle - from Prototypes to Efficient Implementations, in: S. Budkowski, et.al.: 1st International Workshop of the Formal Description Technique Estelle (ESTELLE'98), Evry, France, Nov. 1998
13. XTP Forum, Xpress Transport Protocol Specification, XTP Rev. 4.0, XTP Forum, Santa Barbara, USA, 1995